SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 241–256 (1996)

Research

An Exploratory Study of Common Coding Faults in C Progams

ISHBEL DUNCAN AND DAVE ROBSON
Department of Computer Science, University of Durham, Durham, DH1 3LE, U.K.

SUMMARY

Large scale code testing can be mad viable by determining and searching for the most probable faults in the system under examination. The results of an exploratory survey carried out to determine the common error factors for code written in C indicates that the program task and the programmer experience are important considerations. Using this information, testing can be directed efficiently towards the removal of prevalent coding faults. The results of the survey can also be useful in determining features of C which are likely to become the subject of corrective maintenance.

KEY WORDS: testing; metrics; C language; reliability

INTRODUCTION

Software is becoming increasingly complex in terms of length and module interaction. Testers rarely have enough time and resources to facilitate a full examination of newly developed code because of frequent schedule slippage in the project development. An existing system, altered for user demand or environmental changes is difficult to test systematically, invariably lacking specifications, target requirements or simply information on usage. Testing tools either check for error subsets, such as data flow errors or unreachable code, ascertain statement coverage or aid test data generation. Few research test tools have been executed on industrial or commercial sized software. These tools are often best suited to small programs in order to generate meaningful results and require experienced testers to execute them and understand the complex output. The approach taken in this paper centres on searching for probable errors in the software under test. Finding the statistically most likely errors may not ensure a completely robust system, but should give some positive assurance of a well-founded test. As the test progresses, less habitual faults can be searched for, implying a decreasing error find rate. Testing can consume up to 50 per cent of the project time (Lientz and Swanson, 1980), but is rarely allocated this much time. Testers must then enhance their techniques to uncover likely faults before directing themselves towards the more subtle, but perhaps, pernicious faults. The more likely errors in any system may depend on several factors such as the language or operating system used, the programmer experience, company development practices or simply the task the system is attempting to undertake.

CCC 1040–550X/96/040241–16

Received 1991
Revised 29 April 1996

The authors investigated the use of mutation analysis as a test technique for checking large scale programs (Duncan and Robson, 1993). In mutation analysis, a component in the source code is altered in some way to simulate a fault, for example, an identifier is initialized to a value different from the original. Test data inputs are then executed on the changed, or mutant, program and the output compared with the original output. If the existing test data yield identical output in both cases, the data are considered inadequate. Test data adequacy refers to the ability of data to differentiate a correct program from its incorrect mutants (Acree *et al.*, 1979; Duncan, 1994) and must be strengthened to cause the mutant to output incorrectly. As a technique, mutation analysis dictates the development of adequate test data, and also provides a rigorous check on the code. However, a major problem with mutation analysis concerns its resource consumption; alterations to all operands, operators and even statement reordering or deletion require large storage capacity and fast compilation, execution and comparative routines. Present mutation analysis tools (DeMillo *et al.*, 1988; Woodward, 1991), allow mutants to be stored in single line descriptions applied to low level executable code, or use vector processing architecture to enable high speed processing (Krauser, Mathur and Rego, 1988). More recent mutation analysis tools focus on weak mutation analysis tools, where comparisons are made locally to the alteration (Sahinoglu and Spafford, 1990). Users are also provided with a choice of mutant subsets, that is, a tester can choose to enable say, all the operator mutations or all variable definition alterations. However, these subsets do not take the common problems of the language or other contributing factors into account. To develop mutation analysis as an efficient tool, applicable to large programs, it is proposed to simulate the most prevalent fault initially. If time and resources allow, all other fault simulations should be completed. In order to categorize the most likely problems in programs written in C, chosen because of its wide use in the academic and commercial world, an exploratory survey of programmers and testers of C code was undertaken.

It was also anticipated that the preparatory study would reveal problems which are likely to become the subject of corrective maintenance. Although corrective maintenance is not the most expensive contributor to maintenance costs (Lientz and Swanson, 1980), it still forms a significant portion. The identification of likely future problems can clearly lead to the adoption of strategies which will minimize the likelihood of these problems.

THE SURVEY

A questionnaire was developed based on personal experience on the C language and a notable publication concerning problems encountered in C code (Koenig, 1989). The 40 interviewees were chosen in equal numbers, 20 from three academic science departments and 20 from a large national and commercial organization. Within each institution, the candidates were chosen with regard to their proven abilities in design, programming, maintenance or testing of code. All were science graduates and had research and/or industrial experience. The academics were computer science research staff, specializing in research in software testing or software maintenance, computer centre staff, who maintain and develop systems for general university use and a group of physicists who build fine tooled instruments and use software written in C to drive the tools. The non-academics were professional testers, programmers or systems designers. All interviewees had programmed in C and updated code, not necessarily their own code and all had at least

several years experience of coding in various languages. The non-academics comprised around half of all the candidates. Novice C users, those with less than two years experience, also accounted for half the candidates. All candidates had knowledge of at least one other language.

The candidates were interviewed individually and the interviewer completed the questionnaire. Most of the survey involved the interviewee ranking a specific fault as being persistent, occasional, infrequent or non-existent in occurrence. The bias inherent with regard to the type of work, ability and forthrightness of the interviewee was accounted for by preliminary questions regarding work and experience. Later questioning prompted the candidate to discuss frequently occurring faults without reference to the earlier ranked questions. This section attempted to encourage the interviewee to discuss particularly prevalent problems associated with their programming or maintenance tasks and was used as a cross-reference to the earlier sections.

The objective of the survey was to identify what the candidates thought were the most common faults found in C programs. This would give guidance to fault simulation in a later experiment (Duncan and Robson, 1993). The survey was not, owing to time and research constraints, an in-depth study incorporating the recording of all software problems. It was an exploratory look at problems encountered by professionals.

The survey was divided into six sections (the survey questionnaire is in Appendix II). Section A established the experience of the interviewee and their preferred compiler and operating system. Questions were also asked with respect to personal definition of program trouble spots or critical regions, that is, the most frequently used segments or modules, error handling routines, etc. These were naturally biased towards the candidate's most recent experiences, but it was expected that some information on test approach methods could be obtained. Interviewees were requested to indicate their habitual method of test data generation from random methods, code coverage, output class coverage (equivalent output classes) or path coverage schemes.

The following three sections centred on specific problems in source code semantics and syntax, library and macro usage and in portability issues. Answers were ranked between one and four depending on the error frequency. These questions applied to all software the interviewee had worked with, of their own writing or not.

The last section queried the interviewee's bias or sensitivity to the most regular errors found at compilation or execution time. The candidate was also asked to indicate any particular constructs for which they programmed defensively, that is, code formations the programmer or tester took care in developing or examining. Candidates also indicated the most frequent causation of faults found after testing was ostensibly completed. They indicated problems as being logical, semantic, constructional or specificational in origin. Some further questions summarized aspects of mutation analysis by querying whether the interviewees had encountered some common faults applied as mutations by mutation analysis tools.

RESULTS

In the following sections, the strongest statistical relationships are discussed. There are many others which appear weak and may be due to non-linear or compound relationships or may be specific to any peculiarities in the data. It should be noted that these results are based on the programmer's or tester's perception of an error in code under test; he

or she may be sensitized to particular faults by the nature of their work. They may ignore or fail to notice faults which they did not expect to find. The interviewee may also be referring to recent code problems and have ignored past problems although asked to summarize all remembered faults.

As the survey data are mainly ranked in nature, only the basic statistical tests such as frequency, multiple response, cross tabulation and non-parametric correlation were applicable (Ehrenberg, 1975). This was considered acceptable because of the nature of the study. A more detailed, long term survey of specific faults and problem conditions would allow more elaborate statistical tests to be undertaken.

GENERAL METHODS AND BACKGROUND

Every candidate answered to C not being their first learned language, 66 per cent had a Pascal or Modula-2 background. Over 50 per cent were Sun/BSD C users, the rest mainly using Microsoft C. Over half checked, tested or used software not developed by themselves.

Asked to indicate one or more reasons for terminating code testing (see Figure 1), 34 per cent of respondents mentioned time constraints and 75 per cent went by presumption of output correctness. This latter designation incorporated those who considered the test program robust after several correct outputs had been viewed, without reference to any coverage metrics or oracles. That is, once the system 'appeared' to output correctly on several outputs, it was deemed correct. Only 11 per cent overall mentioned the use of an error threshold mechanism. The more experienced C users tended to be constrained by time. Comparing these terminating conditions with the most used method of test data generation, it was found that code coverage methods of generation tended to be chosen by those constrained by time or by those who only considered output correctness. Answer driven test data generation, that is, ensuring all output classes are tested, tended to be chosen by those candidates who completed testing when they felt the program was robust rather than following a strict metric. Those who preferred to generate test data from path driven techniques indicated that testing would be finished only after all the possible executable paths had been exhausted. This group believed testing to be a never ending task.

The interviewees were asked to categorize various code modules as critical or non-critical for unit testing purposes under the constraint that time was limited. This was a very general categorization and is non-specific to program task or structure. Testing the

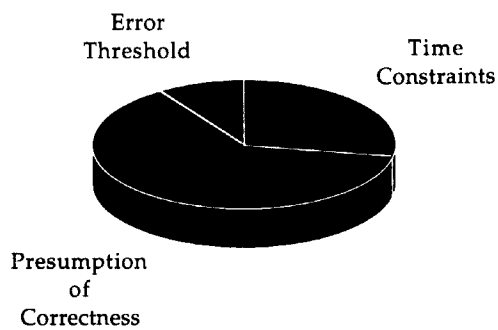

Figure 1. Reasons for terminating testing

most used module was seen as critical by 58 per cent of all respondents and by more of the commercial and experienced C users than others.

40 per cent of respondents categorized the least used module as critical to test under time restrictions, but the more experienced C users ranked it as more vital than others. Error handling and recovery routines were ranked highly, that is, important, by 40 per cent, with the computer scientists considering it as low priority. Special case handling, but not specifically error recovery, was ranked highly by 64 per cent but less by the commercial programmers. End product driven testing, that is checking for coverage of all the output classes, was ranked highly by 72 per cent overall with most of the academic service personnel considering this as very important. Responses indicated that over 60 per cent of all candidates would test modules in which constructs were used which were personally considered troublesome. Those who are more experienced in C considered the testing of error prone constructs in the language to be more critical than other interviewee groups. It was noted that the more experienced programmers consider testing of the whole program, internal modules and interactions as important. This was in contrast to the novices who tended to test distinct logical segments.

With regard to general methods of test data generation, 50 per cent consider random data generation to be the least important technique. However, the physical engineers categorized this as their most common method. Output class techniques, that is, checking that each output as defined by the requirements was reachable, were chosen least by the commercial programmers who preferred code coverage. The academic service personnel mostly chose output class coverage, that is, checking that all partitions of output were reachable. These results match expectation; services have to provide software which performs for the majority of users and commercial companies desire, but do not necessarily achieve, stricter testing than that which is usually contained in research projects. There was little variation in preferred technique with respect to experience, the choice of method varies with programmer or program task classification.

SPECIFIC CODING FAULTS

Candidates ranked replies to questions on error frequency between one and four. A rank of one implied the interviewee had never met or known about the possible fault and a rank of two declared they were aware of it. Ranks of three and four indicated that the candidate had seen an associated fault occasionally or persistently. Appendix I summarizes the results from this section of the survey.

Less than 20 per cent of respondents had difficulty with lexical problems such as the semantic meaning of $++p^*$ or $a=-1$. However, the non-academic testers saw this as a persistent problem in their code, as did those who used the default warning flag on compilation. Some 70 per cent considered confusion of the = and == as a persistent problem. Confusion of the Boolean operators was declared troublesome by the novice C programmers, but overall some 70 per cent deemed this problem infrequent. Over 40 per cent overall regarded operator precedence mistakes as a major problem with more of the commercial candidates considering it frequent. Order of evaluation of the increment operator, as in $++p$ or $p++$, was ranked low except by the computer scientists who thought it was an occasional problem. The computer scientists also had problems with the associativity of operators. More than 60 per cent of respondents thought wrongly placed semicolons was a common fault, but only 35 per cent overall had noticed a fault

generated from a semicolon causing execution of an empty block as in *if (a);*. Many remarked that they actively used this construct. Missing breaks in switches were thought troublesome mainly by computer scientists, but overall there was an even division between those considering this as a common fault and those who had not met the problem. Half of the respondents, and mainly the more experienced, ranked the problem of dangling else's as low. This is the problem of association of an *else* with the most recent *if then* construct and not with any prior outer level *if then*. Those using high warning compilation flags had little difficulty, but those using the default warning had a marked problem. Missing argument lists in a function call was ranked an infrequent problem by 67 per cent of respondents, but confusion of arrays and pointers was ranked highly by over 70 per cent. High warning compilation flags appear to reduce this latter problem. Computer scientists appear to have problems in passing arrays but have less of a problem with array bounds. However, non-academic programmers and experienced C programmers rank array bound problems as occasional or persistent in frequency. Integer wraparounds and overflows are ranked infrequent by over 90 per cent of respondents, but those that did rank it highly wrote mathematical or low level code. Over 70 per cent of interviewees had never known a problem with a return from a main program being unset. Breaks in loops were ranked low by 82 per cent and no one admitted using *goto*. Programs displaying the wrong logic, that is with an incorrect call sequence or statements typed in the wrong order was considered problematic by over 75 per cent of candidates and more especially by the commercial programmers and those with more C experience.

Programs consisting of multiple files or calling libraries formed another section of the questionnaire. The commercial programmers considered the mismatch of definition and declaration and missing initial values to be a persistent problem. Missing type definitions occurred frequently for novice C users and less so for the more advanced user. Programmers used to several different compilers had persistent problems with type definitions. More than one variable definition and function declaration confusion was regarded as equally frequent or infrequent. Name conflicts between files was not seen to be a problem, except by the commercial and the more experienced programmers. Function result type confusion is a problem for 60 per cent of respondents but argument type confusion is troublesome for only half. Few people had difficulty with the use of the *errno* and *signal* functions, but those who remarked on problems were all low level systems programmers. *Printf* and *scanf* formats were considered troublesome by half of the respondents.

The use of macros was, in general, considered trouble free. On portability issues, function definition alterations were ranked troublesome by 43 per cent of programmers and a similar figure had difficulty in distinguishing identifier names between systems. This was especially a problem for the commercial programmers. Integer to pointer implementation dependencies were problematic for only 32 per cent of respondents and 54 per cent had difficulty with changes in integer size. Less than 40 per cent had encountered faults in characters being signed or unsigned and only 20 per cent declared difficulties in using the shift operator. Access to memory location zero was troublesome for 40 per cent of candidates and some 60 per cent complained of differences in the use of *malloc* and *realloc* between systems.

The last section of the survey was again general and an attempt to categorize the sensitivity to certain errors by the respondents. This was done for cross reference with earlier replies and to indicate bias. As before, it should be remembered that respondents will be inclined towards problems recently met. When asked to generalize about the main

cause of faults discovered after testing was ostensibly completed, over 50 per cent of respondents indicated a logical failure and nearly 40 per cent said it was a requirements or specificational error. Naming common compilation errors, 25 per cent mentioned typing mistakes, 22 per cent indicated missing or extra semicolons and slightly less said pointer or brace placement problems. Execution errors centred mainly on pointers, short arrays, memory management and semicolon faults.

The interviewees were asked to name any language construct that they programmed defensively around, that is, anything they took care with when coding. Answers included memory management (45 per cent) and operator precedence (24 per cent). Other responses included avoiding the use of the language terseness when possible, checking function returns and avoiding pointer chains. Many mentioned the failure or lack of comments as a problem in maintenance. When asked to give advice on testing code, candidates again mentioned the use of comments as an aid. Initialization was regarded as frequently lacking. Further remarks involved actively using two phase testing procedures, that is, unit then integration testing, implying that these methods are not commonly used. Stress testing was also advocated as was the avoidance of cutting and pasting between programs to upgrade systems.

The last series of questions were based on mutation analysis. Interviewees were asked if they had encountered any of the conditions usually simulated by mutation tools. The use of the wrong identifier had been noticed by 65 per cent of respondents, but 74 per cent had not seen a problem with an incorrect definition of a constant. The use of the wrong relational operator was viewed by 75 per cent of respondents but only 22 per cent had noticed a numeric variable with the wrong sign. Some 56 per cent had encountered a parenthesis fault, that is, a failure associated with precedence, and 75 per cent had seen braces placed in the wrong position, terminating a sequence incorrectly.

SUMMARY

The interviewees were questioned as to the frequency of finding certain errors or faults when coding or testing C programs. They were also asked to generalize about their methods of testing without particular regard to their present work. Some answers were highly specific and task orientated, others were general.

All candidates had prior knowledge of another language. When testing, few used a metric for deciding when to stop. Those that did were professional testers. Many were constrained by time or decided to stop on the basis of several correct outputs being generated.

The majority of interviewees considered the most used module and special case routines as being important to examine even if time was short. Many mentioned checking that all output classes could be generated and examining any routines which contained troublesome language constructs. Few chose to test the least used modules and the error handling routines, but, this should be seen as task specific. For example, a life critical system should be able to recover from compound errors and not abort.

Test data generation techniques differed between the subject groups. The physical engineers used random test data generation methods, the commercial testers preferred code coverage and the service academics chose output class coverage.

The majority of coding errors centred on confusion of the equivalence and assignment operators, semicolons wrongly placed and pointer and array confusions. For larger programs

and portability issues, function type confusion, integer size and the memory allocation routines were all considered troublesome. General logic problems, necessarily specific to the language, were frequently mentioned. These concerned faults with the use of the wrong identifier or relational operator. Also mentioned were precedence problems from badly placed parenthesis and logic termination faults from misplaced braces.

The more experienced C users appeared to have problems with array handling, pointer chains and general logic. As a group, they were more constrained by time, implying perhaps, that they are much in demand and divide their time between several tasks. The commercial programmers and testers complained of problems with array bounds, and, with multiple file systems, of faults in missing initial values, declaration and definition mismatches and identifier conflicts and distinction. This is possibly intuitive given that commercial and industrial programmers work in groups on a project. Interactions between modules written by different coders is well known to be troublesome (Brooks 1980). It should also be stated that the commercial programmers were using much larger systems written by several programmers, whereas the academics were using smaller programs. The academic groups displayed less problems with multiple files and portability faults because, as a group, they rarely code with partners or upgrade software.

Few interviewees encountered problems with integer overflows and shift operator problems. Similarly, few had seen problems associated with the error flagging or asynchronous routines. This is because only a small percentage of those interviewed wrote mathematical or system tasking software. It appears that the majority of the faults can be simulated by mutation analysis and determined at either the unit or integration level. As mutation analysis is largely automatic, it is therefore a powerful technique for testing large systems under development.

RELATED WORK

Hatton (1994) indicated that the equality and assignment operators cause frequent confusion. He studied 54 packages comprising a total of more than one million lines of C. He also indicated that wrongly placed semicolons (at the end of a condition) cause the following bracketed code sections to be always executed. This occurs frequently. Unused variables and non-initialization of variables also have high frequency but he rates these problems as of low or moderate severity (Hatton, 1994, chapter 4). Hatton refers to problems associated with operator precedence and order to evaluation and order of side-effect. The latter two are unspecified in Standard C. He encourages the use of static tools to detect such dependencies and also states that of all statically detectable faults, some 25.8 per cent lie in the interface between functions.

In chapter 2, Hatton (1994) lists 97 explicitly undefined items in Standard C, 76 implementation defined features and 22 unspecified items. Many of the undefined items involve functions (definitions and calls), pointers, library functions and conversion specifications for the *fprintf* and *fscanf* functions. The functions *fprintf* and *fscanf* account for 11 items in Hatton's list of explicitly undefined items in Standard C.

Hatton also mentions the use of compiler optimization and warning flags and agrees that code should be tested with and without optimization. Results should then be compared for equality (Hatton, 1994, chapter 7). He also suggests that once faults are found in a function, the tester should look at clone functions (Hatton, 1994, chapter 7). A clone function is one which has the same functional complexity or number of external variables,

for example. This is comparable with the authors' finding that it is worth looking for faults of the same type or testing a problematic function for other categories of error.

Spuler (1994) discusses general forms of error in C (and C++) including off-by-one, recursion and aliasing problems (Spuler, 1994, chapter 12). Further chapters list common errors in tokens, expressions, flow of control and declarations (Spuler, 1994, chapters 13–16) as well as library, memory and preprocessor faults (Spuler, 1994, chapters 18–29). No mention is made of commonality or frequency but it is noted that Spuler names confusion of equality and assignment, logical operators, operator precedence, order of evaluation and data flow anomalies (including non-initialization). Problems with the library functions *scanf* and *printf* are also severally mentioned.

FOLLOWING WORK

The survey indicated that many faults are simple in origin, although it may be that the most dangerous of pernicious faults cannot be modelled by mutation analysis. A prototype mutation testing system was built to induce simple faults in C code. This system, called the Grail (Duncan and Robson, 1993), analysed multi-function code by performing simple mutations on up to 11 different components.

Several programs were analysed, the largest being over 1 800 lines of code, comprising nearly 40 functions. The results showed that although mutations duplicate simple faults, that is, the simple alteration of an operator or variable identifier for example, many are difficult to detect. This is perhaps owing to clustering of interdependent faults or error masking by subsequent code execution. Mutation analysis also demands 100 per cent statement coverage as a minimal requirement for defect detection. In some tests, notably the larger programs, some mutations died under application of a test case, but came back to life in another test. The mutants were termed Zombie mutants and were due to memory garbage problems. The mutation operator which mostly demonstrated the Zombie mutants was the assignment operator. The statement '$a = 0$' could be mutated to '$a+ = 0$' in C. The mutant could therefore be either live or dead depending on the value stored in the memory location accessed by variable 'a'. This enforced initialization of variables to solve the problem of Zombie mutant programs.

CONCLUSION

It is difficult to judge whether errors are noticed and eradicated because of experience or sensitivity to particular faults. One is biased towards the latter scenario as experience dictates greater knowledge of the language and its usage. Once a fault is discovered, testers tend to look for more of the same kind and as such become wary of that particular problem. Given this natural bias, it is still apparent that the work designation of the interviewees had some effect on the errors they considered troublesome. It can be argued that this is to do with in-house practices and test techniques but a stronger case is that the program task is a good indicator of likely faults. A scientist writing code to drive machinery, or a mathematician, is more likely to meet integer size problems when developing and porting software than a software engineer creating an editor. A commercial coder who works in a large team will more frequently have problems with parameter passing or distinguishing identifier names than a computer scientist who writes single file programs.

The common method of test data generation differs between the programming groups interviewed. Physicists are more prone to random data generation, service personnel to output class coverage and commercial programmers to code coverage techniques. Experienced programmers view the whole program as the test object more so than novices who tend to look at specific units.

Error proliferation relates to task type, but there is some indication that it is also related to developmental methods. The commercial interviewees found problems with multiple file and library usage that others did not. This may be due to the sheer size of the code developed but may also be related to the number of programmers on the project. In the academic environment it is rare to find more than one person designing and writing code.

The survey indicates some error groupings depending on the programmer's experience and on the program task. Large scale software testing must take these factors into account by searching for the most common errors to give some measure of reliability.

In order to remove the candidate's sensitivity to recent errors found in code, it is considered that a future survey should take a different form. A larger group of experienced candidates would be asked to log all faults and to describe any problems encountered over a long period of time. The resultant data would be more specific and suitable for more thorough statistical tests.

A possible criticism of the methodology of this survey is that it addresses programmers' beliefs about software problems, rather than the problems themselves. There is a relationship between those beliefs and the problems, but in order to reduce or possibly eliminate the difference between these two factors, it would be necessary to independently log all the programmers' faults and problems, which would be a much larger task.

APPENDIX I

This is a summary of ranked answers to errors found in C programs. A rank of 1 indicates that the interviewee had not encountered any problems associated with the construct mentioned. A rank of 2 implies they were aware of problems and ranks of 3 or 4 indicate that the problem was encountered occasionally or persistently.

Problematic Construct	Rank Frequency			
	1	2	3	4
Lexical problems	10	17	8	2
Confusion of & and &&	14	12	7	3
Confusion of ! and	15	15	5	1
Order of evaluation of ++	13	15	8	1
Missing breaks in switch	13	7	13	4
Missing argument in func call	14	9	9	5
Passing arrays to functions	17	12	7	2
Reading into arrays with ++	15	12	9	1
General use of goto	31	5		
Control error	14	3	11	3
Confusion of = and ==	4	6	19	8
Confusion of | and ||	15	15	3	3

Operator precedence	7	14	14	2
Operator associativity	12	18	7	
Dangling else	18	7	9	3
Confusion of pointer with array	3	9	14	11
Array bounds	9	10	12	6
Integer overflow	22	10	3	2
Breaks in loops	21	8	7	1
General wrong logic	3	5	18	5
Multiple File Usage				
Mismatch of defn and declaration	12	2	17	5
Missing initialization	7	9	12	9
Confusion of funct decl types	12	6	13	4
Function result confusion	10	5	15	6
Getchar returns	18	9	6	4
Errno function usage	15	5	5	1
Printf with wrong types	9	11	12	5
Buffering problems	15	3	8	4
Missing type definition	16	5	13	3
More than one definition	13	4	15	5
Name conflicts	10	6	9	2
Funct argument type confusion	10	6	11	9
Sequential file usage	27	2	5	2
Signal function usage	25	5	4	2
Scanf with wrong types	8	12	9	8
Portability Issues				
Function definition	8	6	9	4
Integer to pointer dependencies	11	8	7	3
Chars signed or unsigned	10	6	7	5
Access to memory location 0	14	3	7	4
Random number generation	21	4	2	
Memory allocation	8	4	10	6
Printf formats	16	5	7	
Identifier distintion	11	5	7	5
Size of integers	9	6	8	7
Shift operator problems	11	7	2	1
Division truncation	21	4	2	1
Case conversion routines	20	3	4	
Nested comments	15	5	7	1

APPENDIX II

The survey

The following questions attempt to ascertain the most common programming errors in the C language and whether they are specification, programmer or version dependent. The assumption is that the program under test compiles without error.

Part A. Experience of interviewee

1. How many years have you been programming in C?
2. How many years have you been programming?
3. Do you check/test other people's software?
4. If you are in charge of tasking, do you put more experienced programmers on certain types of software? If so,
   Experienced on type?
   Novice on type?
5. How do you, if necessary, define the critical regions of your program, i.e., those parts of the program you feel should be thoroughly tested?
   (Rank individually as 1—not at all, 5—extremely critical)
   Most used block/module?
   Least used block/module?
   Error handling?
   Exception Handling?
   End product driven? (i.e., test all types of output)
   Personal common mistakes? (i.e., areas of code with problem constructs)
   Knowledge of error prone constructs in C?
6. How do you generate your own test data?
   (Rank group as 1 to 4: 1—most important, 4—least important)
   Random?
   Path driven? (i.e., all possibilities of segment connection)
   Answer driven? (i.e., all output ranges covered)
   Code coverage driven? (i.e., all segments covered at least once)
7. What do you consider adequate code coverage/when would you stop testing?
8. What language(s) would you choose if C was not available?
9. What experience do you have in that language? (years)
10. What version of C do you most use?
11. What version of C do you prefer?
12. What o.s. and version do you most use with C?
13. What version of o.s. do you prefer with C?

Section B. Source construction errors

14. Are any of the following a problem in any program you have worked on (your

own or anothers)?
(Rank each as
1—not at all
2—aware of problem
3—occasional problem
4—persistent problem)
Lexical munching? (e.g., $a = -1$; $*p++$)
Confusion of = and ==?
Confusion of & and &&?
Confusion of | and ||?
Confusion of ! and ¬?
Operator precedence?
Order of evaluation of increment operator?
Order of evaluation of operators?
Semicolons in wrong place?
Force an empty block?
Missing before function? (main)
Missing breaks in switch?
Dangling else?
Missing argument list in function call?
Confusion of pointer with data array?
Confusion of data array with pointer?
Passing array to a function passes address of first element?
Array bounds?
Reading into arrays using ++?
Integer overflow with signed integers?
Integer wraparound with unsigned integers?
Return from main unset?
Breaks in loops?
GOTOs?
Wrong logic (statements positioned wrongly)?
Control errors (use of wrong control structure)?

Section C. Linkage and library problems

15. Are any of the following a problem in any program you have worked on (your own or anothers)?
(Rank each as
1—not at all
2—aware of problem
3—occasional problem
4—persistent problem)

Confusion of declarations and definitions?
Mismatch of definition and declaration?

Missing type definition?
Missing initial value?
More than one definition?
Confusion of function declaration types?
Name conflicts? (length or case of identifier)
Result type confusion?
Argument type confusion?
Getchar returns an integer?
Sequential file update?
Call to errno?
The signal function?
Calls to printf with wrong types?
Calls to scanf with wrong types?
Buffering problems?

Section D. Macro usage problems

16. Are any of the following a problem in any program you have worked on (your own or anothers)?
(Rank each as
1—not at all
2—aware of problem
3—occasional problem
4—persistent problem)

Spaces in macros?
Parenthesis in marcros?
Confusion of macros with functions?
Problems with the use of macros for type definitions?

Section E. Portability problems

17. Are any of the following a problem in any program you have worked on (your own or anothers)?
(Rank each as
1—not at all
2—aware of problem
3—occasional problem
4—persistent problem)

Problems with function definitions?
Distinguishing identifier names?
Conversion of integer to pointer is implementation dependent?
Size of integers?

Signed or unsigned characters?
Shift operator problems?
  Vacated bits zeroed or copies of sign?
  Values for shift count?
Memory location zero?
Division truncation?
Random number generation?
Case conversion?
Reallocation of memory?
8 and 9 as octal digits?
Nested comments?
Printf formats?

Section F. Programmer's bias

18. What do you 'feel' are the most common errors made in programming in C:
    By yourself
      Compilation errors?
      Run-time testing?
    By others
      Compilation errors?
      Run-time testing?
19. Favourite anecdote?
20. Do your own errors change as you become more experienced?
      What errors as novice?
      What errors as expert?
21. What compiler warning level do you now use in development?
22. What problem areas do you actively program around or program defensively for (which constructs do you take time with)?
23. Do problem areas change with product?
    If so, what is the most common in
      DBMS?
      Machine control?
      Network control?
      Other?
24. If, after testing is finished, a fault is discovered, what is the most common problem? Is it logical, semantic, constructional, requirements or specificational?
25. Do you use C++? If so, what for?
26. Would you be willing to submit a single file, medium sized source file for testing? Do you have any data for this?
27. Any other points or suggestions?

Acknowledgements

The authors wish to acknowledge the assistance of our commercial sponsors, and staff and researchers at the University of Durham. They also wish to thank the anonymous referees for their helpful comments.

References

Acree, A. T., Budd, T. A., Lipton, R. J., DeMillo, R. A. and Sayward, F. G. (1979) 'Mutation analysis', Technical Report GIT-ICS-90/08, School of Information and Computer Science, Georgia Institute of Technology, Altanta, GA.

Brooks, F. P. (1980) *The Mythical Man Month*, Addison-Wesley, Reading, MA, USA.

DeMillo, R. A., Guindi, D. S., King, K. N., McKracken, W. M. and Offutt, A. J. (1988) 'An extended overview of the Mothra software testing environment', in *Proceeding 2nd Workshop on Software Testing, Verification and Analysis*, IEEE Computer Society Press, Banff, Alberta, pp. 142–151.

Duncan, I. M. M. and Robson, D. J. (1993) 'Mutation analysis and the GRAIL system', in *Proceedings of Pacific North-West Software Quality Conference*, Portland, Oregon.

Duncan, I. M. M. (1994) 'Strong mutation testing strategies', Ph.D. Thesis, University of Durham.

Ehrenberg, A. S. C. (1975) *Data Reduction*, Wiley-Interscience, London.

Hatton, L. (1994) *Safer C: Developing Software for High-integrity and Safety-critical Systems*, McGraw-Hill, London.

Krauser, E. W., Mathur, A. P. and Rego, V. (1988) 'High performance testing on SIMD machines', in Proceedings 2nd Workshop on Software Testing, Verification and Analysis, IEEE Computer Society Press, Banff, Alberta.

Koenig, A. (1989) *C Traps and Pit Falls*, Addison-Wesley, Reading, MA, USA.

Lientz, B. P. and Swanson, E. F. (1980) *Software Maintenance Management*, Addison-Wesley, London.

Sahingoglu, M. and Spafford, E. H. (1990) 'Sequential statistical procedures for approving test sets using mutation-based software testing', Technical Report SERC-TR-79-P, Purdue University, West Lafayette, Indiana 47907.

Spuler, D. A. (1994) *C++ and C Debugging, Testing and Reliability*, Prentice Hall, Sydney.

Woodward, M. R. (1991) 'Concerning ordered mutation testing of relational operators', *Journal of Software Testing, Verification and Reliability*, **1** (3), 35–40.

Authors' biographies:

Ishbel Duncan is currently a senior research assistant in the Department of Computer Science at the University of Durham, researching into testing and metrics. She worked at the University of Technology, Sydney on object orientated metrics and testing before returning to Durham where she had studied for a Ph.D. on large scale mutation testing under Dave Robson. Previously she worked as a university programming adviser, a college lecturer in robotics and information technology, and as a mathematics teacher in a secondary school.

Dave Robson is a Lecturer in Computer Science and Sub-Dean of the Faculty of Science at the University of Durham. His areas of research are software testing and the human aspects of software development. He has been previously employed at the Universities of Hull and Southampton and also at the University of Victoria, Canada. He is a Chartered Engineer and a Fellow of the British Computer Society.